**Theoretical Analysis:**

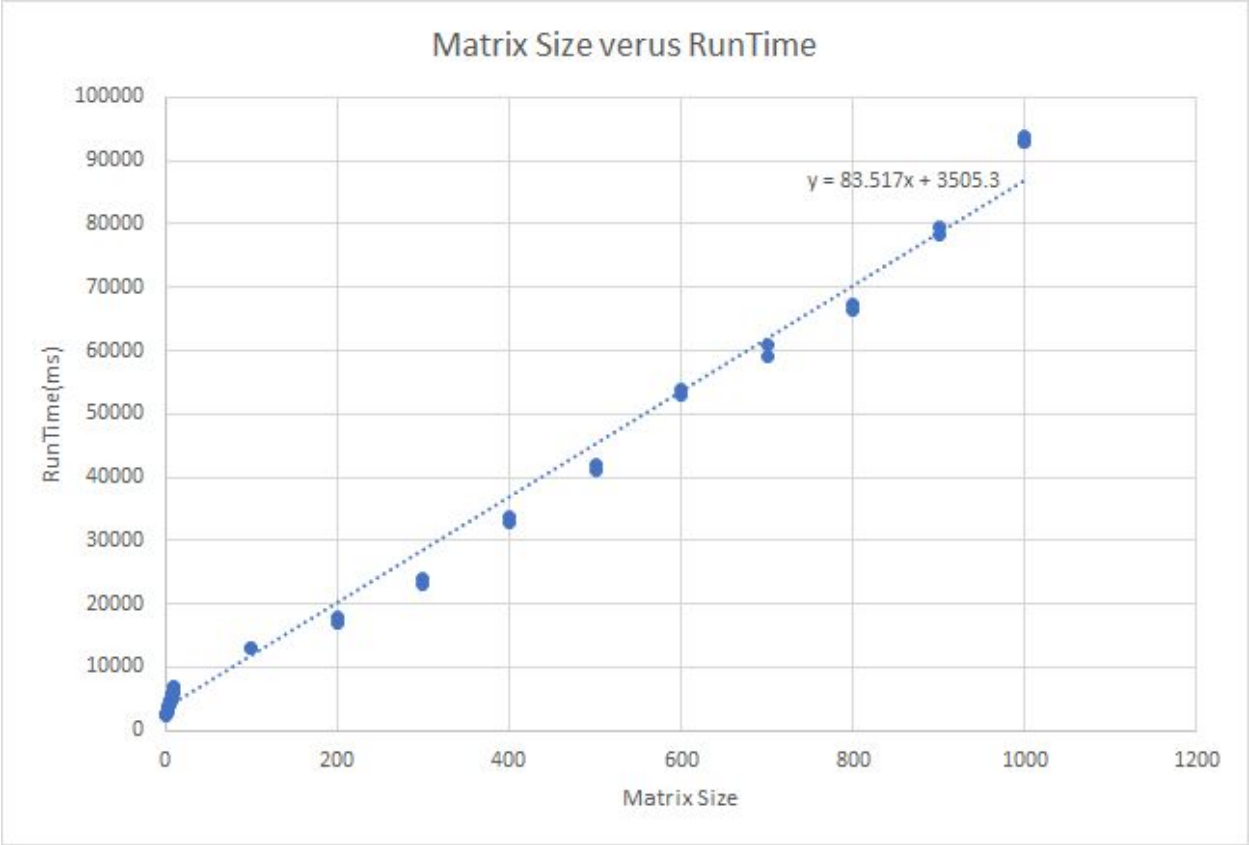
By analyzing my inMatrix method code an O(n) runtime is seen. This is because when analyzing the inMatrix method, I saw by summing the time-complexity statements from each line we get:: O(1)+ O(1)+O(1)+ O(1) + O(1) +(O(2N) * O (1))) => O(2N). Since, constants can be ignored we get the total time complexity to be => O(N). This calculation linear time complexity and analysis of each statement is outlined in the comments of the code in bold shown below.

```
public static boolean inMatrix(int[][]matrix, int x)
    {
        if(matrix == null || matrix.length == 0 || matrix[0].length == 0) //O(1)
            return false;  //simple statement that has a constant complexity = O(1)

            int n = matrix.length; //simple statement that has a constant
                                   //complexity = (Number of rows)=> O(1)
        int m = matrix[0].length; //simple statement that has a constant
                                  //complexity = (Number of columns)=> O(1)

        int i = 0; //simple statement thus has a constant complexity = O(1)
        int j = m - 1; //simple statement thus has a constant complexity =O(1)

        // start from top right corner of the matrix
        while (i < n  && j >= 0) { // O(N) + O(N) => O(2N)=> O (N) ((Where N is the
                                   //number of rows and N is the number of columns of
                                  //matrix and N+N = 2N). We can disregard the 2, since
                                  in the long run //constants can be ignored

            if (matrix[i][j] == x) { //simple statement thus has a constant
                                     //complexity = O(1)
                return true;  //simple statement thus has a constant complexity
                              //=O(1)
            } else if (matrix[i][j] > x) {  //simple statement thus has a constant
                                            //complexity =O(1)
                j--; // move left //simple statement thus has a constant
                   //complexity =O(1)
            } else {
                i++; // move down simple statement thus has a constant
                     //complexity =O(1)
            }

        }

        // element not found
        return false; //simple statement thus has a constant complexity = O(1)
    }
```

Analyzing my code theoretically shows a linearly growing time complexity. This is more efficient than if it was growing exponentially, but it is not the most eddicent, as the most efficient would be an algorithm that would solve this with a O(logn) time complexity.

**Empirical Analysis:**

| **Matrix Size** | **RunTime(ms )**(***Note I have 3 numbers separated by commas since I ran each matrix three times --explained more in paragraph under graph***) |
|---|---|
| 1x1 | 2400,2300, 2700 |
| 2x2 | 3000, 3100, 2900 |
| 3x3 | 3700, 3400, 3800 |
| 4x4 | 4200, 4400, 4300 |
| 5x5 | 4500, 4300, 3900 |
| 6x6 | 4900, 5000, 4700 |
| 7x7 | 5200, 5400, 5100 |
| 8x8 | 5600, 5500, 5700 |
| 9x9 | 6200, 6500, 6300 |
| 10x10 | 6900, 6700, 6800 |
| 100x100 | 13100,13300,13600 |
| 200x200 | 18000, 20800, 19000 |
| 300x300 | 24001, 26600, 27100 |
| 400x400 | 35700, 36000, 32000 |
| 500x500 | 48500, 46500, 47000 |
| 600x600 | 56400,57500, 56000 |
| 700x700 | 62200, 61000, 621000 |
| 800x800 | 66300, 65000, 64500 |
| 900x900 | 78400, 78000, 77500 |
| 1000x1000 | 93900, 92800, 92900 |




I inputted the values from the chart above into a graph and got the value below. Through empirically analyzing over time, I saw a linear relationship, big-o of O(n), as predicted while theoretically analyzing my code. As the matrix size increases the run time linearly increases with it. Although there were some outliers with the run time, that was due to threads etc, overall a linear relationship between matrix size and run time is seen. It would have been more efficient if the inMatrix method I wrote was growing logarithmically, hence O(logn), since as the matrix size increases, the time would level off instead of increasing, my program is still more efficient than one with an exponential relationship between matrix size and run time. I fit a straight line,since my data points were linearly varying and used the LINEST function in excel to find that the uncertainty in the slope was 1.08652. Hence, meaning that my points overall displayed an accurate linear relationship between the matrix size and runtime. In my table I have the run times in the table that came from running my algorithm when searching for a number in the bottom corner. Since my algorithm started at the top right corner of the matrix, the farthest value away from it was at the bottom corner, and hence it was the worst-case scenario that I tested the runtimes for, to ensure that I was able to get the time it took when the worst scenario value was imputed. I ran it multiple times to get a variety of data-points, to reduce error in analysis.